HIGH-LEVEL DATA RACES

Cyrille Artho and Armin Biere
Computer Systems Institute
ETH Zurich, Switzerland
{*artho,biere*}*@inf.ethz.ch*

Klaus Havelund
Kestrel Technology
NASA Ames Research Center,
Moffett Field, California USA
havelund@email.arc.nasa.gov

Key words: Java, multi-threading, data races, software verification, consistency

Abstract: Data races are a common problem in concurrent and multi-threaded programming. They are hard to detect without proper tool support. Despite the successful application of such tools, experience shows that the notion of data race is not powerful enough to capture certain types of inconsistencies occurring in practice. In this paper we investigate data races on a higher abstraction layer. This enables us to detect inconsistent uses of shared variables, even if no classical race condition occurs. For example, a data structure representing a coordinate pair may have to be treated atomically. By lifting the meaning of a data race to a higher level, such problems can now be covered. The paper defines the concepts *view* and *view consistency* to give a notation for this novel kind of property. It describes what kinds of errors can be detected with this new definition, and where its limitations are. It also gives a formal guideline for using data structures in a multi-threaded environment.

1 INTRODUCTION

Multi-threaded, or concurrent, programming is becoming increasingly popular in enterprise applications and information systems (Artho and Biere, 2001; Sun Microsystems, 2002). The Java programming language (Arnold and Gosling, 1996) explicitly supports this paradigm (Lea, 1997). Multi-threaded programming, however, provides a potential for introducing intermittent concurrency errors that are hard to find using traditional testing. The main source of this problem is that a multi-threaded program may execute differently from one run to another due to the apparent randomness in the way threads are scheduled. Since testing typically cannot explore all schedules, some bad schedules may never be discovered. One kind of error that often occurs in multi-threaded programs is a *data race*, as defined below. In this paper we shall go beyond the traditional notion of what we shall refer to as low-level data races, and introduce high-level data races, together with an algorithm for detecting them. The algorithm has been implemented in the Java PathExplorer (JPaX) tool (Havelund and Roşu, 2001) (Goldberg and Havelund, 2002) (Bensalem and Havelund, 2003), which provides a general framework for instrumenting Java programs, and for monitoring and analyzing execution traces. In particular JPaX contains algorithms for detecting problems in multi-threaded programs, such as data races and deadlocks (Bensalem and Havelund, 2003). Although JPaX analyzes Java programs, the principles and theory presented here are universal and apply in full to concurrent programs written in languages like C and C++ as well (Nichols et al., 1998).

1.1 Low-level Data Races

The traditional definition of a data race is as follows (Savage et al., 1997):

> A data race occurs when two concurrent threads access a shared variable and when at least one access is a write, and the threads use no explicit mechanism to prevent the accesses from being simultaneous.

Consider for example two threads, that both access a shared object containing a counter variable *x*, and assume that both threads call an *increase()* method on the object, which increases *x* by 1. The *increase()* method is compiled into a sequence of bytecode instructions (load *x* to the operand stack, add 1, write back the result). The Java Virtual Machine (JVM) executes this sequence non-atomically. Suppose the two

threads call *increase()* at nearly the same time and that each of the threads execute the *load* instruction first, which loads the value of *x* to the thread-local operand stack. Then they will both add 1 to the original value, which results in a combined increment of 1 instead of 2. We shall refer to this traditional notion of data race as a *low-level data race*, since it focuses on a single variable.

The standard way to avoid low-level data races on a variable is to protect the variable with a lock: all accessing threads must acquire this lock before accessing the variable, and release it again after. In Java, methods can be defined as `synchronized` which causes a call to such a method to lock the current object instance. Return from the method will release the lock. Java also provides an explicit statement form `synchronized(`*obj*`)`{*stmt*}, for taking a lock on the object *obj*, and executing statement *stmt* protected under that lock. If the above mentioned *increase()* method is declared `synchronized`, the low-level data race cannot occur.

Several algorithms and tools have been developed for analyzing multi-threaded programs for low-level data races. The Eraser algorithm (Savage et al., 1997), which has been implemented in the Visual Threads tool (Harrow, 2000) to analyze C and C++ programs, is an example of a dynamic algorithm that examines a program execution trace for locking patterns and variable accesses in order to predict potential data races. The Eraser algorithm maintains a *lock set* for each *variable*, which is the set of locks that have been owned by all threads accessing the variable in the past. Each new access causes a refinement of the lock set to the intersection of the lock set with the set of locks currently owned by the accessing thread. The set is initialized to the set of locks owned by the first accessing thread. If the set ever becomes empty, a data race is possible. Another commercial tool performing low-level data race analysis is JProbe (Sitraka, 2000).

1.2 High-level Data Races

A program may be inconsistent even when it is free of low-level data races, where we consider the set of locks protecting a single variable. In this paper we shall turn this around and study the *variable set* associated to a *lock*. This notion makes it possible to detect what we shall refer to as *high-level data races*. The inspiration for this problem was originally due to a small example provided by Doug Lea (Lea, 2000). It is presented in modified form in Sec. 2. It defines a simple class representing a coordinate pair with two components x and y. All accesses are protected by synchronization on `this`, using `synchronized` methods. Therefore, data race conditions on a low level are not possible. As this example will illustrate, there can still be data races on a higher level, and this can be detected as inconsistencies in the granularity of *variable sets* associated to locks in different threads. The algorithm for detecting high-level data races is a dynamic execution trace analysis algorithm like the Eraser algorithm (Savage et al., 1997).

As a realistic example of a high-level data race situation, we shall illustrate a problem that was detected in NASA's *Remote Agent* space craft controller (Pell et al., 1997). The problem was originally detected using model checking, as described in (Havelund et al., 2001). The error was very subtle, and was originally regarded hard to find without actually exploring all execution traces as done by a model checker. As it turns out, it is an example of a high-level data race, and can therefore be detected with the low-complexity algorithm presented in this paper.

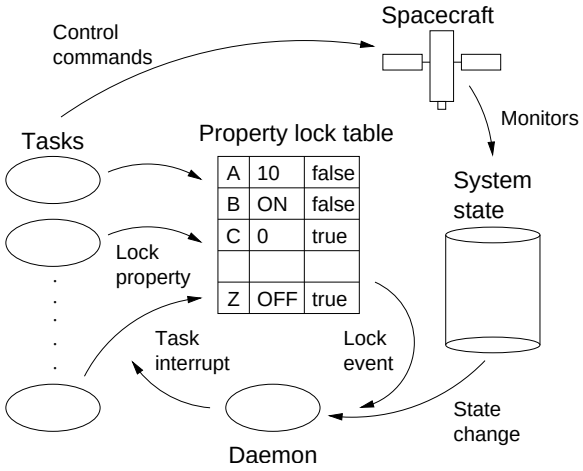

Figure 1: The Remote Agent Executive

The Remote Agent is an artificial-intelligence-based software system for generating and executing plans on board a space craft. A plan essentially specifies a set of tasks to be executed within certain time constraints. The plan execution is performed by the *Executive*. A sub-component of the Executive is responsible for managing the execution of tasks, once the tasks have been activated. The data structures needed for managing task execution are illustrated in Fig. 1. The state of the spacecraft (at any particular point) can be considered as an assignment of values to a fixed set of variables, each corresponding to a component sensor on board the space craft. The spacecraft maintains a current *system state*. The term *property* is used to refer to a particular assignment for a particular variable. A task may require that specific properties hold during its execution. Upon the start of the task, it first tries to lock those properties it requires in a *lock table*. For example, a task may require *B* to be *ON*. Now other threads cannot request *B* to be *OFF* as long

Task

```
synchronized (table) {
  table[N].value = V;
}

/* achieve property */

synchronized (table) {
  table[N].achieved = true;
}
```

Daemon

```
synchronized (table) {
  if (table[N].achieved &&
    system_state[N] != table[N].value) {
    issueWarning();
  }
}
```

Figure 2: The synchronization inconsistency between a task and the daemon.

as the property is locked in the lock table. Next, the task tries to achieve this property (changing the state of the space craft, and thereby the system state), and when it is achieved, the task sets a flag *achieved* to *true* in the lock table, which has been *false* until then.

A *Daemon* constantly monitors the lock table, and checks that: *if a property's flag* achieved *is true, then it must be a true property of the space craft, and hence true in the system state*. Violations of this property may occur by unexpected events on board the space craft. The daemon wakes up whenever events occur, such as when the lock table or the system state are modified. In case an inconsistency is detected, the involved tasks are interrupted.

The relevant code from the task and the daemon is illustrated in Fig. 2, using Java syntax (The Remote Agent was coded in LISP). The task contains two separate accesses to the lock table, one where it updates the value and one where it updates flag *achieved*. The daemon on the other hand accesses all these fields in one atomic block. This can be described as an inconsistency in lock views, as described below, and actually presents an error potential.

The error scenario is as follows: suppose the task has just achieved the property, and is about to execute the second synchronized block, setting flag *achieved* to true. Suppose now however, that suddenly, due to unpredicted events, the property is destroyed on board the space craft, and hence in the system state, and that the daemon wakes up, and performs all checks. Since flag *achieved* is false, the daemon reasons incorrectly that the property is not supposed to hold in the system state, and hence it does not detect any inconsistency with the lock table (although conceptually now there is one). Only then the task continues, and sets flag *achieved* to true. The result is that the violation has been missed by the daemon.

Detecting this error using normal testing is very hard since it requires not only to execute the just described interleaving (or a similar one), but it also requires the formulation of a correctness property that can be tested for, and which is violated in the above scenario. However, regarding this as a view inconsistency problem allows us to find the error without actually executing this particular interleaving, and it does not require a requirement specification. The view inconsistency in this example can be described as follows:

> The daemon accesses the value and flag *achieved* in one atomic block, while the task accesses them in two different blocks. Hence, the daemon has view $v_1 = \{value, flag\}$ while the task has views $v_2 = \{value\}$ and $v_3 = \{flag\}$. This is view-inconsistent since the task's views form disjoint subsets of the daemon view.

This view inconsistency is in itself not an error. However, in the above example it is a symptom that if pointed out may direct the programmer's attention to the real problem, that property achievement and setting flag *achieved* are not done in one atomic block[1]. More formal and generic definitions of view inconsistency are presented in Sec. 2 and 3.

1.3 Outline

The paper is organized as follows. Sec. 2 introduces the problem of high-level data races. Sec. 3 presents the concepts for detecting high-level data races. The JPaX framework for analyzing Java programs is described in Sec. 4. Sec. 5 describes the experiments carried out. Sec. 6 gives an overview of related work. Sec. 7 outlines future work and Sec. 8 concludes the paper.

2 INTUITION

Consistent lock protection for a shared field ensures that no concurrent modification is possible. However,

[1]Note that repairing the situation is non-trivial since achieving properties may take several clock cycles, and it is therefore not desirable to hold the lock on the table during this process.

```
class Coord {
  double x, y;
  public Coord(double px, double py) { x = px; y = py; }
  synchronized double getX() { return x; }
  synchronized double getY() { return y; }
  synchronized Coord getXY() { return new Coord(x, y); }
  synchronized void setX(double px) { x = px; }
  synchronized void setY(double py) { y = py; }
  synchronized void setXY(Coord c) { x = c.x; y = c.y; }
}
```

Figure 3: The `Coord` class encapsulating points with x and y coordinates.

Thread t_1

```
synchronized (c) {
  access(x);
  access(y);
}
```

Thread t_2

```
synchronized (c) {
  access(x);
}
```

Thread t_3

```
synchronized (c) {
  access(x);
}
synchronized (c) {
  access(y);
}
```

Thread t_4

```
synchronized (c) {
  access(x);
}
synchronized (c) {
  access(x);
  access(y);
}
```

Figure 4: One thread using a pair of fields and three other threads accessing components individually.

this only refers to low-level access to the fields, not their entire use or their use in conjunction with other fields. The remainder of this paper assumes detection of low-level data races is covered by the Eraser algorithm (Savage et al., 1997), which can be applied in conjunction with our analysis.

Fig. 3 shows a class implementing a two-dimensional coordinate pair with two fields x, y, which are guarded by a single lock. If only `getXY`, `setXY`, and the constructor are used by any thread, the pair is treated atomically. However, the versatility offered by the other accessor (`get`/`set`) methods is dangerous: if a thread only uses `getXY` and `setXY` and relies on complete atomicity of these operations, threads using the other accessor methods may falsify this assumption.

Imagine a case where one thread reads both coordinates while another one sets them to zero. If the write operation occurs in two phases, `setX` and `setY`, the other thread may read an *intermediate result* which contains the value of x already set to zero but still the original y value. This is clearly an undesired and often unexpected behavior. We will use the term *high-level data race* to describe this kind of scenario.

Nevertheless, there exist scenarios where some of the other access methods are allowed and pair-wise consistency is still maintained. The novel concept of *view consistency* captures this notion of consistency while allowing partial accesses. In previous work (Savage et al., 1997), only the use of *locks for each variable* has been considered. The opposite perspective, the use of *variables under each lock*, is the core of our new idea.

Fig. 4 shows another example with four threads, which is abbreviated for better readability. Reading and writing are abstracted as `access(f)`, where `f` is a shared field. Calls of synchronized methods offering access protection are represented using `synchronized(lock){access(f);}` as an abstraction of the inlined method. Thread creation is not shown. Control structures within threads are hidden as well. Furthermore, it is assumed that each field accessed by a thread is a reference to a shared object, visible to all threads.

Initially, we only consider the first two threads t_1 and t_2. It is not trivial to see whether an access conflict occurs or not. As long as t_2 does not use y as well, it does not violate the first thread's assumption that the coordinates are treated atomically. Even though t_1 accesses the entire pair $\{x, y\}$ atomically and t_2 does not, the access to x alone can be seen as a partial read or partial write. A read access to x may be interpreted as reading $\{x, y\}$ and discarding y; a write access may be seen as writing to x while leaving y unchanged. So both threads t_1 and t_2 behave in a consistent manner.

Each thread is allowed to use only a part of the coordinates, as long as that use is consistent. Inconsistencies might arise with thread t_3, which uses x in one operation and y in another operation, releasing

the lock in between. If, for example, thread t_3 *reads* its data in two parts, with another thread like t_1 writing to it in between, t_3 may obtain partial values corresponding to two *different* global states. If, on the other hand, thread t_3 *writes* its data in two parts, other threads, like t_1, may read data corresponding to an *intermediate* state.

Since both read and write accesses result in an error, we do not have to distinguish between the two kinds of access operations, assuming that shared values are not read-only. The difficulty in analyzing such inconsistencies lies in the wish to still allow partial accesses to sets of fields, like the access to x of thread t_2.

As an example of a situation which at first sight appears to provide a conflict, but which we shall regard as safe, consider the situation between t_1 and t_4. This could potentially be regarded as a conflict. However, observing t_4, the second synchronization statement is completely self contained, and accesses in addition to y everything the first synchronization statement accesses (x). Consequently, the first synchronization statement in t_4 likely represents an operation that does not need y (whether read or write). Therefore, the two synchronization operations are unrelated and can be interleaved with the atomic synchronization statement in t_1 without interfering with the operations of t_4 on x and y.

On a more formal basis, t_4 is safe because the set of variables accessed in the first synchronization statement of t_4 is a subset of the set of variables accessed in its second synchronization statement. Put differently, the variable sets form a *chain*. Generally, a set F of fields of a thread t is atomic if they are accessed in a synchronization statement in t. A high-level data race occurs when a thread has an atomic set of fields F and another thread has atomic sets G_1 and G_2 such their overlaps with F do not form a chain. This will be formalized in the next section.

3 VIEW CONSISTENCY

This section defines *view consistency*. It lifts the common notion of a data race on a single shared variable to a higher level, covering sets of shared variables and their uses.

3.1 Views

A lock *guards* a shared field if it is held during an access to that field. The same lock may guard several shared fields. Views express what fields are guarded by a lock. Let I be the set of object instances generated by a particular run of a Java program. Then F is the set of all fields of all instances in I.

A *view* $v \in \mathbb{P}(F)$ is a subset of F. Let l be a lock, t a thread, and $B(t,l)$ the set of all `synchronized` blocks using lock l executed by thread t. For $b \in B(t,l)$, a view *generated by* t with respect to l, is defined as the set of fields accessed in b by t. The *set of generated views* $V(t) \subseteq \mathbb{P}(F)$ of a thread t is the set of all views v generated by t. In the previous example in Fig. 4, thread t_1 using both coordinates atomically generates view $v_1 = \{x, y\}$ under lock $l =$ `c`. Thread t_2 only accesses x alone under l, having view $v_2 = \{x\}$. Thread t_3 generates two views: $V(t_3) = \{\{x\}, \{y\}\}$. Thread t_4 also generates two views: $V(t_4) = \{\{x\}, \{x, y\}\}$.

3.2 Views in Different Threads

A view v_m generated by a thread t is a *maximal view*, iff it is maximal with respect to set inclusion in $V(t)$:

$$\forall v \in V(t)\ [v_m \subseteq v \rightarrow v_m = v]$$

Let $M(t)$ denote the set of all maximal views of thread t. Only two views which have fields in common can be responsible for a conflict. This observation is the motivation for the following definition. Given a set of views $V(t)$ generated by t and a view v' generated by another thread, the *overlapping views* of t with v' are all non-empty intersections of views in $V(t)$ with v':

$$\mathrm{overlap}(t, v') \equiv \{v' \cap v \mid (v \in V(t)) \wedge (v \cap v' \neq \emptyset)\}$$

A set of views $V(t)$ is *compatible* with the maximal view v_m of another thread iff all overlapping views of t with v_m form a chain:

$$\mathrm{compatible}(t, v_m) \quad \text{iff}$$
$$\forall v_1, v_2 \in \mathrm{overlap}(t, v_m)\ [v_1 \subseteq v_2 \vee v_2 \subseteq v_1]$$

View consistency is defined as mutual compatibility between all threads: A thread is only allowed to use views that are compatible with the maximal views of all other threads.

$$\forall t_1 \neq t_2,\ v_m \in M(t_1)\ [\mathrm{compatible}(t_2, v_m)]$$

In the example in Fig. 4, we had

$$V(t_1) = M(t_1) = \{\{x, y\}\}$$
$$V(t_2) = M(t_2) = \{\{x\}\}$$
$$V(t_3) = M(t_3) = \{\{x\}, \{y\}\}$$
$$V(t_4) = \{\{x\}, \{x, y\}\}$$
$$M(t_4) = \{\{x, y\}\}$$

There is a conflict between $t_1$and t_3 as stated, since $\{x, y\} \in M(t_1)$ intersects with the elements in $V(t_3)$ to $\{x\}$ and $\{y\}$, which do not form a chain. A similar conflict exists between t_3 and t_4.

#		Thread t_1	Thread t_2	Incompatible views
1	Views $V(t)$ Maximal views $M(t)$	$\{x\}, \{y\}$ $\{x\}, \{y\}$	$\{x\}, \{y\}$ $\{x\}, \{y\}$	none
2	Views $V(t)$ Maximal views $M(t)$	$\{x, y\}$ $\{x, y\}$	$\{x\}, \{y\}$ $\{x\}, \{y\}$	$\{x\} = \{x, y\} \cap \{x\} \in M(t_1) \cap V(t_2)$ $\{y\} = \{x, y\} \cap \{y\} \in M(t_1) \cap V(t_2)$
3	Views $V(t)$ Maximal views $M(t)$	$\{x, y\}, \{x\}, \{y\}$ $\{x, y\}$	$\{x\}, \{y\}$ $\{x\}, \{y\}$	$\{x\} = \{x, y\} \cap \{x\} \in M(t_1) \cap V(t_2)$ $\{y\} = \{x, y\} \cap \{y\} \in M(t_1) \cap V(t_2)$
4	Views $V(t)$ Maximal views $M(t)$	$\{x, y, z\}$ $\{x, y, z\}$	$\{x, y\}, \{x\}$ $\{x, y\}$	none

Table 1: Examples with two threads illustrating the principle of view consistency.

#		Thread t_1	Thread t_2	Thread t_3	Incompatible views
5	$V(t)$ $M(t)$	$\{x, y\}$ $\{x, y\}$	$\{x\}$ $\{x\}$	$\{x\}, \{y\}$ $\{x\}, \{y\}$	$\{x\} = \{x, y\} \cap \{x\} \in M(t_1) \cap V(t_3)$ $\{y\} = \{x, y\} \cap \{y\} \in M(t_1) \cap V(t_3)$
6	$V(t)$ $M(t)$	$\{x, y\}$ $\{x, y\}$	$\{x\}$ $\{x\}$	$\{y\}$ $\{y\}$	none
7	$V(t)$ $M(t)$	$\{x, y\}, \{x\}, \{y\}$ $\{x, y\}$	$\{y, z\}, \{y\}, \{z\}$ $\{y, z\}$	$\{z, x\}, \{z\}, \{x\}$ $\{z, x\}$	none
8	$V(t)$ $M(t)$	$\{x, y\}, \{x\}, \{y, z\}$ $\{x, y\}, \{y, z\}$	$\{y, z\}, \{y\}, \{z\}$ $\{y, z\}$	$\{z, x\}, \{z\}, \{x\}$ $\{z, x\}$	$\{y\} = \{y, z\} \cap \{y\} \in M(t_1) \cap V(t_2)$ $\{z\} = \{y, z\} \cap \{z\} \in M(t_1) \cap V(t_2)$

Table 2: Examples with three threads illustrating the principle of view consistency.

The above definition of *view consistency* uses three concepts: the notion of *maximal views*, the notion of *overlaps*, and finally the *compatible* notion, also referred to as the *chain* property. The chain property is the core concept. Maximal views do not really contribute to the solution other than to make it more efficient to calculate and reduce the number of warnings if a violation is found. The notion of overlaps is used to filter out irrelevant variables.

3.3 Examples

A few examples help to illustrate the concept. Table 1 contains examples involving two threads. Note that the outermost brackets for the set of sets are omitted for better readability. Example 1 is the trivial case where no thread treats the two fields $\{x\}$, $\{y\}$ atomically. Therefore there is no inconsistency. However, if thread t_1 treats $\{x, y\}$ as a pair, and thread t_2 does not, we have a conflict as shown in example 2. This even holds if the first thread itself uses partial accesses on $\{x\}$ or $\{y\}$, since this does not change its maximal view. Example 3 shows that case. Finally, example 4 illustrates the case where thread t_1 uses a three-dimensional coordinate set atomically and thread t_2 reads or updates different subsets of it. Since the subsets are compatible as defined in Sec. 3.2, there is no inconsistency.

Table 2 shows four cases with three threads. The first entry, example 5, corresponds to the first three threads in the example in Fig. 4 in Sec. 2. There, thread t_3 violates t_1's assumption about the atomicity of the pair $\{x, y\}$. Example 6 shows a "fixed" version, where t_3 does not access $\{x\}$. Finally, more complex circular dependencies are thinkable with three threads. Such a case is shown in example 7. Out of three fields, each thread only uses two of them, but they are used atomically. Because the accesses of any thread only overlap with each other's in one field, there is no inconsistency. This example only requires a minor change, shown in example 8, to make it faulty: Assume t_1's third view were $\{y, z\}$ instead of $\{y\}$. This would contribute another maximal view $\{y, z\}$, which conflicts with t_2's views $\{y\}$ and $\{z\}$.

3.4 Soundness and Completeness

Essentially, this approach tries to infer what the developer intended when writing the multi-threaded code, by discovering view inconsistencies. However, an inconsistency may not automatically imply a fault in the software. An inconsistency that does not correspond to a fault is referred to as a *false positive* (spurious warning). Similarly, lack of a reported inconsistency does not automatically imply lack of a fault. Such a missing inconsistency report for an existing fault is referred to as a *false negative* (missed fault).

False positives are possible if a thread uses a coarser locking than actually required by operation semantics. This may be used to make the code shorter or faster, since locking and unlocking can be expensive. Releasing the lock between two independent

operations requires splitting one `synchronized` block into two blocks.

False negatives are possible if all views are consistent, but locking is still insufficient. Assume a set of fields that must be accessed atomically, but is only accessed one element at a time by every thread. Then no view of any thread includes all variables as one set, and the view consistency approach cannot find the problem. Another source of false negatives is the fact that a particular (random) run through the program may not reveal the inconsistent views.

The fact that false positives are possible means that the solution is not sound. Similarly, the possibility of false negatives means that the solution neither is complete. This may seem surprising, but actually also characterizes the Eraser low-level data race detection algorithm (Savage et al., 1997) implemented in the commercial Visual Threads tool (Harrow, 2000), as well as the deadlock detection algorithm also implemented in the same tool. The same holds for the similar algorithms implemented in JPaX. The reason for the usefulness of such algorithms is that they still have a much higher chance of detecting an error than if one relies on actually executing the particular interleaving that leads to an error, without requiring much computational resources. These algorithms are essentially based on turning the property to be verified (in this case: no high-level data races) into a more testable property (view consistency). This aspect is discussed in more detail in (Bensalem and Havelund, 2003) in relation to deadlock detection.

4 ANALYSIS FRAMEWORK

The experiments were all made with JPaX (Havelund and Roşu, 2001) (Goldberg and Havelund, 2002), a run-time verification tool consisting of two parts: an instrumentation module and an observer module. The instrumentation module produces an instrumented version of the program, which when executed generates an event log with the information required for the observer to determine the correctness of the examined properties. Fig. 5 illustrates the situation.

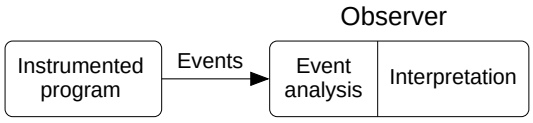

Figure 5: Structure of the run-time analysis: The instrumented program generates a series of events. The observation of these events is divided into two stages: an event analysis and an interpretation of events.

The observer used here only checks for high-level data races. For these experiments, a new and yet totally un-optimized version of JPaX was used. It instruments every field access, regardless of whether it can be statically proven to be thread-safe. Because of this, some data-intensive applications created log files which grew prohibitively large (> 0.5 GB) and could not be analyzed.

4.1 Java Bytecode Instrumentation

Part of JPaX is a very general and powerful instrumentation package for instrumenting Java bytecode (Goldberg and Havelund, 2002). The requirements of the instrumentation package include power, flexibility, ease of use, portability, and efficiency. Alternative approaches were rejected, such as instrumenting Java source code, using the debugging interface, and modifying the Java Virtual Machine because they violated one or another of these requirements.

It is essential to minimize the impact of the instrumentation on program execution. This is especially the case for real time applications, which may particularly benefit from this approach. Low-impact instrumentation may require careful trade-offs between what is computed locally by the instrumentation and the amount of data that need be transmitted to the observer. The instrumentation package allows such trades to be made by allowing seamless insertion of Java code at any program point.

Code is instrumented based on an *instrument specification* that consists of a collection of predicate-action rules. A predicate is a filter on source code statements. These predicates are conjunctions of atomic predicates that include predicates that distinguish statement types, presence of method invocations, pattern-matched references to fields and local variables and so on. The actions are specifications describing the inserted instrumentation code. Actions are inserted where predicates evaluate to true. The actions include reporting the program point (method, and source statement number), a time stamp, the executing thread, the statement type, the value of variables or an expression, and invocation of auxiliary methods. Values of primitive types are recorded in the event log, but if the value is an object, a unique integer descriptor of the object is recorded.

The instrumentation has been implemented using Jtrek (Cohen, 1999), a Java API that provides lower-level instrumentation functionality. In general, use of bytecode instrumentation, and use of Jtrek in particular, has worked out well, but there are some remaining challenges with respect to instrumenting the concurrency aspects of program execution.

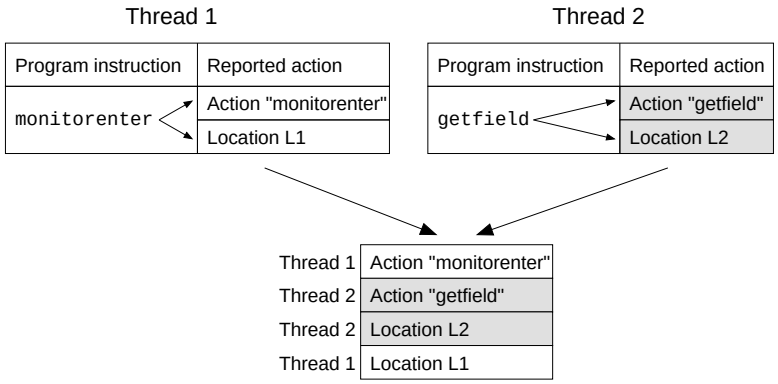

Figure 6: Interleaving events generated by the instrumented program.

4.2 Event Stream Format

All operations in the instrumented application writing to the event log have to be as fast as possible. Among other factors, light-weight locking, incurring as little lock contention as possible, helps achieving this. When several pieces of information are logged by the instrumentation, they are therefore recorded separately, not atomically. As a result of this, one event can generate several log entries. Log entries of different threads may therefore be interleaved. The contextual information, transmitted in *internal events*, include thread names, code locations, and reentrant acquisitions of locks (lock counts). The event analysis package maintains a database with the full context of the event log.

Fig. 6 shows such a scenario: Thread 1 is obtaining a lock, which is done with a `monitorenter` instruction in the Java bytecode. Thread 2 is accessing a field, shown by the `getfield` instruction. Both events have been instrumented to record not only the action itself, but also the exact location in the code where the action took place. The two events can be interleaved because they are not recorded atomically.

In order to allow a faithful reconstruction of the events, each log entry includes the hash code of the active thread creating the log entry. Therefore the events can all be assigned to the original threads.

4.3 Observer Architecture

As described above, run-time analysis is divided into two parts: instrumenting and running the instrumented program, which produces a series of events, and observing these events. The second part, event observation, can be split into two stages: event analysis, which reads the events and reconstructs the runtime context, and event interpretation (see Fig. 7). Note that there may be many event interpreters.

Reusing the context reconstruction module allows for writing simpler event interpreters, which can subscribe to particular event types made accessible through an observer interface (Gamma et al., 1995) and which are completely decoupled from each other.

Each event interpreter builds its own model of the event trace, which may consist of dependency graphs or other data structures. It is up to the event interpreter to record all relevant information for keeping a history of the events, since the context maintained by the event analysis changes dynamically with the event evaluation. Any information that needs to be kept for the final output, in addition to the context information, needs to be stored by the event interpreter. If an analysis detects violations of its rules in the model, it can then show the results using stored data.

Besides clearly separating two aspects of event evaluation, this approach has other advantages: Many algorithms dealing with multi-threading problems require very similar information, namely lock and field accesses. If a log generated by an instrumented program includes at least this information, then several analysis algorithms can share the same events. Furthermore, splitting event observation into two steps also allows writing an event analysis front-end for event logs generated by tools other than JPaX, reusing the back-end, event interpretation.

5 EXPERIMENTS

Four applications were analyzed. Those applications include a discrete-event elevator simulator, and two task-parallel applications: SOR (Successive Over-Relaxation over a 2D grid), and a Travelling Salesman Problem (TSP) application. The latter two use worker threads (Lea, 1997) to solve the global problem. Many thanks go to Christoph von Praun for kindly providing these examples, which were referred

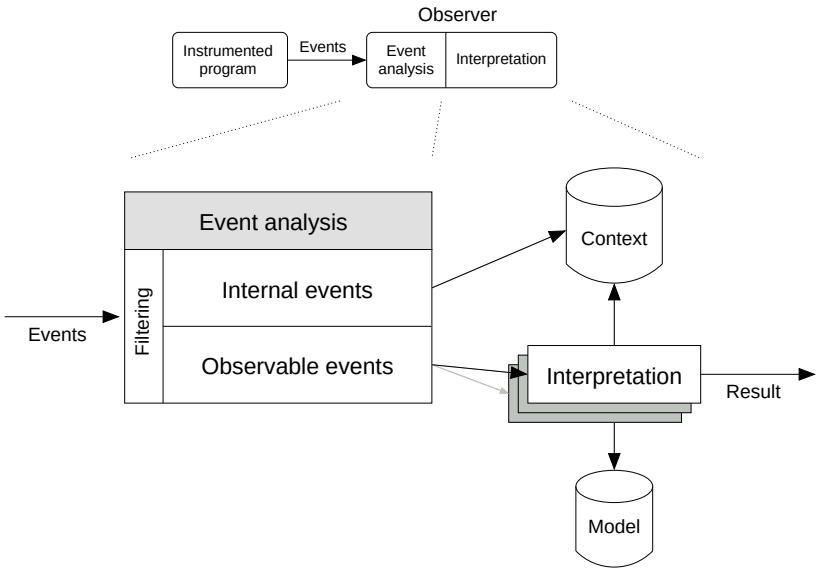

Figure 7: The observer architecture.

Application	Size [LOC]	Run time [s], uninstrumented	Run time [s], instrumented	Log size [MB]	Warnings issued
Elevator	500	16.7	17.5	1.9	2
SOR	250	0.8	343.2	123.5	0
TSP, very small run (4 cities)	700	0.6	1.8	0.2	0
TSP, larger run (10 cities)	700	0.6	28.1	2.3	0
NASA's K9 Rover controller	7000	–	–	–	1

Table 3: Analysis results for the given example applications.

to in (von Praun and Gross, 2001). In addition, a Java model of a NASA planetary rover controller, named K9, was analyzed. The original code is written in C++ and contains about 35,000 lines of code, while the Java model is a heavily abstracted version with 7,000 lines. Nevertheless, it still includes the original, very complex, synchronization patterns.

Table 3 summarizes the results of the experiments. All experiments were run on a Pentium III with a clock frequency of 750 MHz using Sun's Java 1.4 Virtual Machine, given 1 GB of memory. Only applications which could complete without running out of memory were considered. It should be noted that the overhead of the built-in Just-In-Time (JIT) compiler amounts to 0.4 s, so a run time of 0.6 s actually means only about 0.2 s were used for executing the Java application. The Rover application could not be executed on the same machine where the other tests were run, so no time is given there.

It is obvious that certain applications using large data sets incurred a disproportionately high overhead in their instrumented version. Most examples passed the view consistency checks without any warnings reported. For the elevator example, two false warnings referred to two symmetrical cases. In both cases, three fields were involved in the conflict. In thread t_1, the views $V(t_1) = \{\{1,3\},\{3\},\{2,3\}\}$ were inconsistent with the maximal view $v_m = \{1,2,3\}$ of t_2. While this looks like a simple case, the interesting aspect is that one method in t_1 included a *conditional* access to field 1. If that branch had been executed, the view $\{2,3\}$ would actually have been $\{1,2,3\}$, and there would have been no inconsistency reported. Since not executing the branch corresponds to reading data and discarding the result, both warnings are false positives.

One warning was also reported for the NASA K9 rover code. It concerned six fields which were accessed by two threads in three methods. The responsible developer explained the large scope of the maximal view with six fields as an optimization, and hence it was not considered an error. The Remote Agent space craft controller was only available in LISP, so it could not be directly tested. However, we have

successfully applied our tool to test cases reflecting different constellations including that particular high-level data race.

So far, experiments indicate that experienced programmers intuitively adhere to the principle of view consistency. Violations can be found, but are not very common, as shown in our experiments. Some optimizations produce warnings that constitute no error. Finally, the two false positives from the elevator example show that the definition of view consistency still needs some refinement.

6 RELATED WORK

6.1 Static Analysis and Model Checking

Beyond Eraser, several static analysis tools exist that examine a program for low-level data races. The Jlint tool (Artho and Biere, 2001) is such an example. The ESC (Detlefs et al., 1998) tool is also based on static analysis, or more generally on theorem proving. It, however, requires annotation of the program, and does not appear to be as efficient as the Eraser algorithm in finding low-level data races. Dynamic tools have the advantage of having more precise information available in the execution trace. More heavyweight dynamic approaches include model checking, which explores all possible schedules in a program. Recently, model checkers have been developed that apply directly to programs (instead of just on models thereof). For example, the Java PathFinder system (JPF) developed by NASA (Havelund and Pressburger, 2000; Visser et al., 2000), and similar systems (Godefroid, 1997; Corbett et al., 2000; Holzmann and Smith, 1999; Ball et al., 2001; Stoller, 2000). Such systems, however, suffer from the state space explosion problem. In (Havelund, 2000) we describe an extension of Java PathFinder which performs low-level data race analysis (and deadlock analysis) in simulation mode, whereafter the model checker is used to demonstrate whether the data race (deadlock) warnings are real or not. However, a data race, low-level as well as high-level, can be hard to find with model checking since it typically needs to cause a violation of some explicitly stated property.

6.2 Database Concurrency

In database theory, shared data is stored in a database and accessed by different processes. Each process performs *transactions*, sequences of read and write operations, on the data. A sequence of these operations corresponding to several transaction is called a *history*. Based on this history, it can be inferred whether each transaction is *serializable*, i.e., whether its outcome corresponds to having run that transaction in isolation (Papadimitriou, 1979; Bernstein et al., 1987).

There are several parallels to multi-threaded programs, which share their data in memory instead of in a database. Data races on shared fields in a multi-threaded program can be be mapped to database access conflicts on shared records. Lock protection in a multi-threaded program corresponds to an encapsulation of read and write accesses in a transaction. The key problem addressed by this paper, having intermediate states accessible when writing non-atomically a set of fields, maps to the *inconsistent retrieval* problem in databases. In such a history, one transaction reads some data items in between another transaction's updates on these items. A correct *transaction scheduler* will prevent such an access conflict, as long as the accesses of each process are correctly encapsulated in transactions.

High-level data races concern accesses to sets of fields, where different accesses use different sets. Similar problems may be seen in databases, if the programmer incorrectly defines transactions which are too fine-grained. For example, assume a system consists of a global database and an application using reading and writing threads. The writing threads use two transactions to update the database, the reading threads access everything in a single transaction. Here, the reader's view is inconsistent, since it may read an intermediate state of the system. If the writer uses a single transaction, the fault is corrected. We do not have experience whether this is actually an important problem in databases. It is likely that the abstraction provided by database query languages such as SQL (Chamberlin and Boyce, 1976) prevents some of these problems occurring.

Furthermore, concurrency theory as used for databases and transaction systems is moving towards richer semantics and more general operations, called *activities* (Schuldt et al., 2002). Activities are atomic events in such a system. Like in classical transactions, low-level access conflicts are prevented by a scheduler which orders these operations. We are not sure how high-level access conflicts have to be treated with the richer semantics of activities.

Finally, database theory also uses the term *view* under different meanings. Specifically, the two terms *view equivalence* and *view serializability* are used (Bernstein et al., 1987). These two terms are independent of view consistency as defined in this paper.

6.3 Hardware Concurrency

In hardware design and compiler construction, Lamport has made a major step towards correct shared

memory architectures for multiprocessors (Lamport, 1979). He uses *sequential consistency* as a criterion for ensuring correctness of interleaved operations. It requires all data operations to appear to have executed atomically. The order in which these operations execute has to be consistent with the order seen by individual processes.

Herlihy and Wing use a different correctness condition called *linearizability* (Herlihy and Wing, 1990). It provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and response. Linearizability is a stronger property than sequential consistency and has the advantage that it preserves real-time ordering of operations. Although the theory is very general, it is geared towards hardware and compiler construction because it allows exploiting special properties of concurrent objects where transactions would be too restrictive. However, it is not directly applicable to multi-valued objects and seems to be incapable of capturing such high-level problems.

Lamport's notion of sequential consistency is rather restrictive and can be relaxed such that processors are allowed to read older copies of data as long as the observed behavior is indistinguishable from a conventional shared memory system (Afek et al., 1993). Mittal and Garg extended this work and Herlihy's linearizability (Herlihy and Wing, 1990) to multi-object operations, such as double-register compare and swap operations (Mittal and Garg, 1998). Problems occurring with such multi-object operations are very much alike to our high-level data races. Unlike our approach, which deals with access patterns, their approach is concerned with the interleaving of operations and based on histories as known in database literature.

7 FUTURE WORK

There are many areas in which this work can be expanded. They can be classified into technical and theoretical problems.

On the technical side, there are still issues with the run-time analysis tool JPaX. The code instrumentation and event generation does not always provide a reliable identification of objects. It relies on name, type, and hash code of objects. The latter can change during execution, which causes difficulties in the observer. Nonetheless, the hash code is the best identification which can be obtained easily in Java.

Furthermore, the instrumentation has to be optimized with respect to statically provable thread-safety. For instance, read-only or thread-local variables do not have to be monitored. Another optimization would be to only execute logging instructions a few times, instead of every time they are reached. A few executions of each instruction (one by each involved thread) is often enough to detect a problem. Apart from that, the observer analysis could run on-the-fly without event logging. This would certainly eliminate most scalability problems. Additionally, the current version reports the same conflict for different instances of the same object class.

On the theoretical side, it is not yet fully understood how to properly deal with nested locks. The views of the inner locks cause conflicts with the larger views of the outer locks. These conflicts are spurious. The elevator case study has shown that a slightly different, control-flow independent definition of view consistency is needed. Perhaps static analysis may be better suited to check such a revised definition. Finally, we intend to study the relationship to database concurrency and hardware concurrency theory.

8 CONCLUSIONS

Data races denote a concurrent access to shared variables where an insufficient lock protection can lead to a corrupted program state. Classical, or low-level, data races concern accesses to single fields. Our new notion of high-level data races deals with accesses to sets of fields which are related and should be accessed atomically.

View consistency is a novel concept considering the association of variable sets to locks. This permits detecting high-level data races that can lead to an inconsistent program state, similar to classical low-level data races. Experiments on a small set of applications have shown that developers seem to follow the guideline of view consistency to a surprisingly large extent. We think this concept, now formally defined, captures an important idea in multi-threading design.

REFERENCES

Afek, Y., Brown, G., and Merritt, M. (1993). Lazy Caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–205.

Arnold, K. and Gosling, J. (1996). *The Java Programming Language*. Addison-Wesley.

Artho, C. and Biere, A. (2001). Applying Static Analysis to Large-Scale, Multi-threaded Java Programs. In Grant, D., editor, *Proc. 13th ASWEC*, pages 68–75. IEEE Computer Society.

Ball, T., Podelski, A., and Rajamani, S. (2001). Boolean and Cartesian Abstractions for Model Checking C Programs. In *Proc. TACAS'01: Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, Italy.

Bensalem, S. and Havelund, K. (2003). Reducing False Positives in Runtime Analysis of Deadlocks. Submitted for publication.

Bernstein, P. A., Hadzilacos, V., and Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.

Chamberlin, D. and Boyce, R. (1976). SEQUEL: A structured English query language. In *Proceedings of the 1976 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264.

Cohen, S. (1999). Jtrek. Compaq, `compaq.com/java/download/jtrek`.

Corbett, J., Dwyer, M. B., Hatcliff, J., Pasareanu, C. S., Robby, Laubach, S., and Zheng, H. (2000). Bandera: Extracting Finite-state Models from Java Source Code. In *Proc. 22nd International Conference on Software Engineering*, Ireland. ACM Press.

Detlefs, D. L., Rustan, K., Leino, M., Nelson, G., and Saxe, J. B. (1998). Extended Static Checking. Technical Report 159, Compaq Systems Research Center, Palo Alto, California, USA.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Godefroid, P. (1997). Model Checking for Programming Languages using VeriSoft. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, France.

Goldberg, A. and Havelund, K. (2002). A User-Friendly Package for Instrumenting Java Bytecode. Internal report.

Harrow, J. (2000). Runtime Checking of Multithreaded Applications with Visual Threads. In *7th SPIN Workshop*, volume 1885 of *LNCS*, pages 331–342. Springer.

Havelund, K. (2000). Using Runtime Analysis to Guide Model Checking of Java Programs. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 245–264. Springer.

Havelund, K., Lowry, M. R., and Penix, J. (2001). Formal Analysis of a Space Craft Controller using SPIN. *IEEE Transactions on Software Engineering*, 27(8):749–765. An earlier version occurred in the Proceedings of the 4th SPIN workshop, 1998, Paris, France.

Havelund, K. and Pressburger, T. (2000). Model Checking Java Programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381.

Havelund, K. and Roşu, G. (2001). Monitoring Java Programs with Java PathExplorer. In *Proc. First International Workshop on Runtime Verification (RV'01)*, volume 55 of *ENTCS*, pages 97–114, France. Elsevier Science.

Herlihy, M. and Wing, J. (1990). Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492.

Holzmann, G. and Smith, M. (1999). A Practical Method for Verifying Event-Driven Software. In *Proc. ICSE'99, International Conference on Software Engineering*, USA. IEEE/ACM.

Lamport, L. (1979). How to Make a Multiprocessor that Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.*, 9:690–691.

Lea, D. (1997). *Concurrent Programming in Java*. Addison-Wesley.

Lea, D. (2000). Personal e-mail communication.

Mittal, N. and Garg, V. (1998). Consistency Conditions for Multi-Object Distributed Operations. In *International Conference on Distributed Computing Systems*, pages 582–599.

Nichols, B., Buttlar, D., and Farrell, J. P. (1998). *Pthreads Programming*. O'Reilly.

Papadimitriou, C. (1979). The Serializability of Concurrent Database Updates. *Journal of the ACM (JACM)*, 26(4):631–653.

Pell, B., Gat, E., Keesing, R., Muscettola, N., and Smith, B. (1997). Plan Execution for Autonomous Spacecrafts. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1234–1239. Nagoya, Japan.

Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. (1997). Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411.

Schuldt, H., Alonso, G., Beeri, C., and Schek, H.-J. (2002). Atomicity and Isolation for Transactional Processes. *ACM Transactions on Database Systems (TODS)*, 27(1):63–116.

Sitraka (2000). JProbe. `www.sitraka.com/software/jprobe`.

Stoller, S. D. (2000). Model-Checking Multi-threaded Distributed Java Programs. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 224–244. Springer.

Sun Microsystems (2002). *Java 2 Platform Enterprise Edition Specification*. Sun Microsystems. `java.sun.com/j2ee`.

Visser, W., Havelund, K., Brat, G., and Park, S. (2000). Model Checking Programs. In *Proc. ASE'2000: The 15th IEEE International Conference on Automated Software Engineering*. IEEE CS Press.

von Praun, C. and Gross, T. (2001). Object-Race Detection. In *OOPSLA*, pages 70–82. ACM.